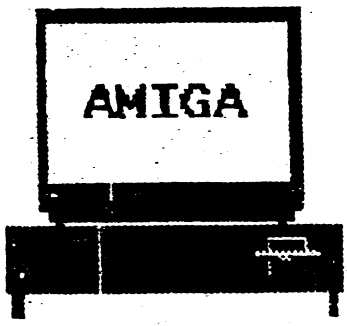
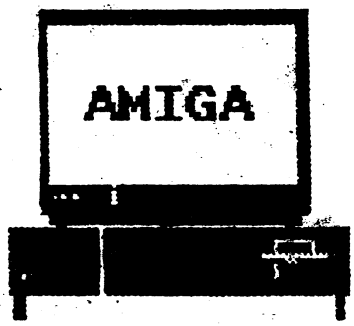


BECAUSE



BULLETIN OF THE CANBERRA AMIGA USERS SOCIETY

MEETINGS:

Meetings are held on the second Wednesday of each month on the second floor of the J.G. Crawford building at ANU, starting at 8:00 PM. Roll up any time after 7.30.

The March meeting will be held on Wednesday the 11th.

SUBSCRIPTIONS:

Annual fees are $20, payable at any of the monthly meetings to the treasurer.

AIMS/BENEFITS:

CAUSe is an independent group formed to bring together people who own, use or are interested in the Commodore Amiga computer. Members receive a bi-monthly bulletin providing a wealth of information concerning the Amiga and are encouraged to attend our monthly meetings.

COMMITTEE:

Director:	Wayne Myles --- 45 8761 (W)	Secretary:	Chris Townley 41 3209 (H)
Treasurer:	John Bishop --- 49 3786 (W)	Editor:	Craig Fisher - 54 6033 (H)
Auxiliary:	Peter Whigham - -- ---- (-)	Auxiliary:	Chris Foster - -- ---- (-)
Auxiliary:	Peter McNeil -- 54 2732 (H)		

BULLETIN BOARD:

CAUSE has it's own FIDO-NET bulletin board which is available free of charge to anyone with a modem.

The board is Baud rate auto-sensing and will accept 300/300, 1200/1200 and 1200/75 baud rates.

Sysop: Mike Hurst-Meyers BBS: 59 1137

IN THIS ISSUE:

Tracker - Workbench utility.

So you think that DISKDOCTOR is great? What about when it tells you to copy all the files that are left onto another disk? - what a chore!

The answer to that chore comes on the WBUTILITY disk in the form of tracker. This program will allow you to copy specified tracks from one disk, in the internal drive to a file, either on disk in another drive or a file in ram. You can then write this file, via the tracker program to another disk, to the same tracks and in the same format. What this means is that by copying all the non-corrupted tracks left by DISKDOCTOR you can effectively copy all your files - ***QUICKLY.***

The utility has other applications, such as copying disks over modem lines by dividing them into files of (say) ten tracks, and then arcing them. At the other end they are simply un-arced and the tracker utility used to make a disk that is the same as the original - no formatting or restructuring.

The utility is used simply by typing **tracker read x y <filename>**, where x and y are track numbers between 0 and 79, and the filename is any valid filename on another device, ie ram:file-1 or df1:file-1. To write to a disk in the internal drive type **tracker write <filename>**, for example **tracker write ram:file-1**, which will write to the tracks read into file-1.

This program seems bug free though a bit sluggish compared to disk copy, although it is quicker than copying by hand. (Understatement). The only complaint would be the use of guilt tactics by "Lord Bradford and the Duke of downloads", (bit of a wank really!); every time you use the program it reminds you that it would really be quite wonderfully nice of you to send $5.00 to Lord Bradford and his mates for doing such a wonderful job of it all. This is all quite right but it does get tedious especially if you send the money.

This is a handy utility which I recommend you try after trashing a disk.

```
      Performance: *****
    Documentation: *****
      Ease of use: *****
Value for money: **** (see article)
 Reality Coeff.: 0.99
   Correctness : *****
```

The Amiga Environment - Part I (Libraries) (Craig Fisher)

This article is aimed at programmers who are not yet familiar with the Amiga programming environment but wish to learn what it offers and how to use it. In this article Amiga Libraries will be covered; Amiga Devices will be described in the next issue of BeCAUSe.

From a user's point of view the Amiga has two different environments - WorkBench and CLI. WorkBench is the Amiga's iconic interface to the filing system and CLI (Command Line Interface) is, as it implies, the command driven interface to the operating system. Most (there may be some who aren't) programmers are more comfortable using the CLI to interact with the Amiga. In fact both the CLI and WorkBench are just programs running on top of the operating system.

It is frequently said that the Amiga hardware is state of the art in design and everyone has seen the wonderful results that it produces, but all those incredible feats could not be performed without the very comprehensive and also state of the art software driving it. The Amiga system software is made up of a hierarchy of modules, each module dedicated to doing different sorts of functions. Writing software on the Amiga is usually at a fairly high level as much of what is to be done is accomplished by a simple call to the appropriate routine in one of the modules. Alternatively, the higher level routines can be bypassed and the machine accessed at a lower level. Basically the Amiga has two types of modules - Libraries and Devices. A library is a collection, or library, of functions for doing specific tasks, and is stored either in ROM or on disk. A device is a software interface to some part of the Amiga hardware.

The standard Amiga libraries are Exec, Clist, Graphics, Layers, Intuition, Mathffp, Mathtrans, MathIEEEdoubbas, Dos, Translator, Icon, Diskfont and Ramlib. Programmers can also put together their own libraries of routines for their programs to use. In order to use a library routine, generally the library must first be opened using the Exec OpenLibrary function. Two of the libraries, Exec and Dos, are normally opened automatically when a program starts up - so their functions are always readily available for use by your programs.

Exec is the most fundamental of all the libraries. It resides in the Amiga's 192K of ROM (presently loaded from KickStart) and provides among other things, the multitasking abilities. All other processes running on the Amiga rely on Exec. It controls the 68000 (CPU), scheduling CPU time between the various tasks running, keeps track of interrupts, starts and stops tasks, and takes care of their priority within the system. Exec also contains routines for managing lists and queues, which are heavily used within the Amiga - almost every structure is stored as part of a linked list (e.g. tasks, screens, windows, libraries, menus, gadgets, files etc). Messages and ports are also handled by Exec for communication between different processes. Input and output is done through devices via a standard from of I/O request provided by Exec. Libraries themselves are also handled by Exec. This library has a whole manual dedicated to it: "ROM Kernel Reference Manual: Exec", written by Commodore-Amiga and published by Addison-Wesley.

The CList library contains a series of routines for manipulating character lists. Unfortunately I haven't yet found out what they are used for or by.

The Graphics library is one of the most significant of all and provides much of the Amiga's magic. The Graphics library does so much that it is hard to give it credit here - whole books have been written about it. Basically there are four types of functions in the graphics library - display, drawing, animation and text. The display related functions do things such as initialising and manipulating bitmaps and bitplanes (places where pictures are stored in memory). The drawing functions cover a range of things from drawing a single line to drawing complex patterned area fills. These functions will also let you do things such as move, copy or complement parts of the display, draw polygons, draw dotted lines or set the screen colours. The animation related functions are also very powerful and allow you to create sprites which can be moved around the display. There are even functions here to handle the colliding of one sprite with another. A sprite's position, velocity and acceleration can all be set and animation then be allowed to run quite automatically. The animation facilities really require an in-depth look on their own some time. Finally there are the text functions which include the handling of fonts and styles (underline, bold, italics) of text.

The Layers library supplies routines for creating and manipulating layers, the name given to a part of memory which holds a 'layer' of a picture. Layers are what makes the Amiga's windowing system possible. Once a layer is created routines from this library can be used to move, size, depth-arrange or delete it. Many of the things which the layers library allows you to do can be accomplished at a slightly higher level through Intuition.

The Intuition library is another very important part of the Amiga system. It relies on the layers and graphics libraries to produce it's displays and so it contains routines which are at a higher level in the Amiga system software hierarchy. Intuition makes programs which make use of it user-friendly by creating and handling screens, windows, menus, requesters, alerts and gadgets. Intuition also makes the handling of input events such as a key being pressed or a mouse button being clicked much easier. I intend to write a series of more in-depth articles about using Intuition in programs, so I won't say anything more about it here.

Mathffp, Mathtrans and MathIEEEdoubbas are, obviously enough, libraries for doing various maths operations. The first, Mathffp, contains routines to do simple fast floating point operations and to convert floating point numbers to and from integers. The Mathtrans library supplies functions to do floating point transcendental operations such as sine, cosine, log etc. The third maths library, with the lovely name of MathIEEEdoubbas contains routines to do basic maths functions such as addition and subtraction using the IEEE format for floating point numbers.

The DOS library holds routines which take care of the manipulation of the filing system. The routines in this library are documented in the AmigaDOS developers manual (this manual is available combined with the other two DOS manuals - the User's Manual and the Technical Reference Manual - in the one book published by Bantam for $39.95). Functions to open, close, read and write files, create, examine and traverse directories and create other processes are included in the DOS library.

The Translator library is used in conjunction with the Narrator device, providing the ability to produce quite good quality speech from within any program. This library in fact only contains a single function, Translate, which converts (translates) an English text string into a stringof phonemes (sound codes) which can then be passed to the Narrator device to be spoken.

The Icon library contains a series of routines to allow programs to manipulate WorkBench icons. These include functions to read, create and write icons associated with tools, projects, drawers etc.

Some programs make use of the Amiga's ability to use a variety of fonts and to do this a special library called the DiskFont library is provided. There are routines here to find out what fonts are available and to open a font.

The last library listed in the Libraries & Devices manual is the Ramlib library although the only information I have been able to find out about this library is that it is of no use when programming in C.

That completes the brief description of the powerful Amiga Libraries. Look in the next issue for a further description of the Amiga Devices.

Happy Amiga'ing...

HANDY HINTS - Textcraft.

Here is something I picked up from Amiga News, Oct 86 about Textcraft "text-only" files, written by Randy Weiner of Commodore.

If you would like to load normal text files from a BBS or text editor into Textcraft, simply copy a one of Textcrafts "text-only" .info files to that files name. For example, to make a README file loadable carry out the following steps.

1. Create a text-only Textcraft file, (It doesn't have to have anything in it). Calling it fred or something.

2. Copy the Fred.info file to README.info,
   ie "copy fred.info README.info"

3. now either boot up with Textcraft, or
   assign sys: "Textcraft 1.0:"

4. then click on the README icon. the README file will now load into Textcraft.

Note, some files may terminate lines with a CR which can be annoying with Textcraft, ie each line will be a new paragraph, however it will still load. This can be fixed by either going through the text file and changing all the CR's to LF's, or going through with Textcraft and backspacing on all the EOL markers.

Programming In C

By J G Bishop

Last issues article dealt with a set of design principles that programs should follow to aid in design, debugging and documenting of code. In this issue we shall begin a short course in programming in the C language. In later issues we shall increasingly examine applying our knowledge of C to programming issues as they relate to the Amiga.

Getting Started

To be C programmers the first thing we need is a copy of Kernighan & Ritchie's Programming in C. This book is the principle reference used to resolve disputes over language issues. Most C compiler writers refer to this book to establish standards, error reporting and usage conventions of language features and routines. Unfortunately it is a bit out of date and does not include the new language features added recently by the C Standards Group. Some of these 'undocumented' features include the void data type, passing structures as parameters and returning structures from routines. While these features are better documented in more recent publications including the report of the C standards committee, K&R remains the preferred reference for the core of the language. Other texts of interest/use include "Handbook of Algorithms and Data Structures" by G.H. Gonnet (Addison-Wesley) and (for the absolute beginners) "A C Primer" by author unknown (I seem to have lent my copy out).

Using Lattice C

If you have followed the install instructions and made a development work disk you will be able to compile and link a C program quite easily.

Compiling

Place the compiler disk in your second drive. If you lack a second drive then compiling and linking will involve lots of disk swaps, or some clever RAM disk handling. A good idea in the latter case is to copy the program to the RAM disk, set this as the current directory and place the compiler disk in the internal drive and run the compiler. Unfortunately this only works for small files and for a larger file you may have to resort to disk swapping. Life is a lot easier if you have a second drive, or a hard disk.

Assume that we have made a file called test1.c using one of the editors. To generate a program from this file we must open a cli, cd to the directory containing test1 follow these steps:

```
1>execute lca test1
```

(This executes the script file in s directory called lca which is a compiler driver call)

The console window will display a few messages, some error or warning messages and eventually the number of modules compiled. If we did not have any errors in test1.c, lca will have generated a file called test1.o which is an object module available for linking. Lattice C allows you to specify multiple files as arguments to the lc compiler driver at a time. If, for example, we had typed:

```
1>"Lattice C:lc" -iINCLUDE: test1 test2 test3
```

We would generate test1.o test2.o and test3.o. Notice that here we used the compiler driver rather than the script file so that we could specify multiple source files. The -i option told the compiler where to find include files not present in the current directory. We will look at include files later. The lca script file in fact looks something like the preceding example.

If test1.c contained the 'main' routine then we would also find a map file generated which is used later by the linking stage.

Lattice C is a Two Phase compiler. This means that it has two phases of the compilation cycle. The first phase :

1) preprocesses the file adding include files to the source stream as if they were part of the original file and expanding defined macros, etc,
2) calculates constant values and performs constant value optimisation (ie 'sizeof(myvariable)' is calculated, and 1+2 is replaced by 3, etc)
3) checks the syntax of your code with any expansions from stage 1, identifying and reporting errors by line number, and
4) generates quads copying these into a quad file (test1.q).

A quad file is a file of 'quads' where a quad has the form:

```
operator argument1 argument2 result
```

This is a kind of machine independent assembler language which is suitable for the kinds of optimisation that Phase 2 of the compiler performs. This second phase consists of :

1) reading the quad file,
2) optimising loops, repeated statements, and if - then - else branches that always evaluate to one branch, etc.
3) generates and writes out from the optimised quad codes the motorolla machine code representation of your C program.

The final step generates the test1.o file which you would subsequently link.

Optimisation is fraught with problems. Not the least of which is that it can easily optimise wrongly. I have only found one optimisation bug in the Lattice C V3.03 compiler. It occurs under
the following circumstances:

```
mybuf[x+1]= 1;
mybuf[x] = 2;
++x;
```

This appears to optimise to:

```
mybuf[++x]=1;
mybuf[x]=2;
```

which is evidently wrong. An easy fix to optimisation errors of this sort is:

```
mybuf[x+1]=1;
mybuf[x] = 2;
DummyProcedure();
++x;
```

The DummyProcedure here is just a call to an empty procedure but the call serves to break the optimisation sequence and correct the error. Most optimisation errors may be controlled in this fashion, even if it is a little untidy.

Syntax errors (errors resulting from incorrect C language 'grammar' usage are, as we have already noted, generated by the first phase of the compiler. So if we wish to a copy of the errors into a file to be printed down, or used by lse (the lattice screen editor) we must redirect the output from the FIRST PHASE of the compilation, NOT from the compiler driver lc. Since lca uses lc to compile programs we can not simply redirect the output from the lca script file. We will need to make an extra script file as follows: (enter your favorite editor and save the following under the name of "s:lce")

```
LC:lc1 ><file$t1>.err -iINCLUDE: -iINCLUDE:lattice/ <file$t1>
```

Now typing:

```
1> execute lce test1
```

will generate a file called test1.err, which will be empty if there were no errors detected in test1. Note the providing lce with test1.c, will cause the errors to be stored under file test1.c.err. Naturally if there were no errors we would need to run lca (or the second phase of the compiler alone) to get an object file for linking since the first phase of the compiler

only generates quad files.

Linking

When we have finally generated test1.o we may link it using the script file linka, provided by lattice (and in your s directory after installation). linka uses alink although this may be replaced with the public domain linker called blink for a faster cleaner link. Linking is the stage where were satisfy inter object file variable and procedure references and link the hunks of code together. The Amiga, unlike most micro computers, does not require its code to be in contiguous memory locations. Instead it allows scatter storage of the hunks of code. In general we may assume that one hunk is equivalent to one module/file of C code generated. The overhead associated with the tables to allow for data and code references when you cant assume an absolute address to jump to, but instead must go indirectly through the index table of the hunk, can substantially add to your code size. This is size may be reduced by unhunking the linked file (unhunk is a public domain utility). The result is a smaller TOTAL file size with one significant snag! The original file was able to be scatter loaded into ram. This means that only the largest contiguous memory block required was as large as the largest module/file in the link. The unhunked version must have contiguous memory as large as the TOTAL file size to load successfully.

Generally you will not need to consider the above facts as the program generated by the linka is usually adequate. The linka

script file has the form:

```
link:alink from LIB:c.o+<file$t1> to <file$t1> lib LIB:lc.lib
     +LIB:amiga.lib map <file$t1>.map
```

(Note the last two lines should be on one line !)

So that a call of the form:

```
1> execute linka test1
```

should cause to be generated a file called "test1" which may be run from the cli OR from an icon you make in IconEdit and name "test1.info". It is worth your while to investigate the code for c.asm which is the Amiga equivalent to the standard C "root" routine which ordinarily starts a C program setting up the file structures, standard IO, processes any arguments provided with the called program and provides them as arguments to the "main(argc,argv)" routine. When main returns (ie the program has completed) it returns to the root which closes any opened level 2 files and tidies up malloc'ed or calloc'ed memory. In the Amiga the c.asm routines are also responsible for determining whether the process was commenced under the workbench from an icon, or from the cli and initialises the argv,argc parameters to main

appropriately. C.asm is in the lib directories on the compiler disk.

Linking more than one module together is not significantly more difficult. Assume test1.c contains the "main()" routine, and test2 and test3 are other modules that we have written that contain functions used by test1, we could link them thus:

```
link:alink from LIB:c.o+test2.o+test3.o+<file$t1> to <file$t1>
     lib lib:lc.lib+LIB:amiga.lib map <file$t1>.map
```

(Again: the last two lines should not be split.)

Next Issue we shall have a brief look at some of the language features and build some machine independent I/O routines, such as simple menu handlers. We shall also have a glimpse of file I/O considerations. Until then....

J.G.Bishop

Console Device Escape Sequences (Craig Fisher)

As some people have already discovered you can change the colours and style of text in the CLI to suit your tastes or to make things stand out. By typing special escape sequences you can set the foreground and background colours of text and set the style to underlined, **bold**, *italics* or any combination of these.

The sequences to use are:

<ESC>[0m	plain text with normal colours
<ESC>[1m	**bold-face** text
<ESC>[3m	*italic* text
<ESC>[4m	underlined text
<ESC>[7m	inverse text (foreground/background colours reversed)
<ESC>[30m to <ESC>[37m	set the foreground colour (note CLI has only four colours)
<ESC>[40m to <ESC>[47m	set the background colour

Note that each escape sequence must end with an "m" (lower case).

More than one attribute can be set in a single escape sequence by using the semicolon ";" as a separator. For example the sequence:

<ESC>[1;32;41m

will turn on the bold style, set the foreground (text) colour to colour 2 (black if using original Workbench colours) and set the background colour to colour 1 (white).

The text style escape sequences are the same as used by the printer device so typing a file with these escape sequences will give the same (or similar) thing on screen as on the printer. The escape sequences can also be entered with an editor such as TxEd, as was done with this article, but you will have to *TYPE* the file to see the result.

PageSetter: Desktop Publishing for Amiga.

First impressions of this program are fairly good; it brings to the Amiga user power of expression that approaches that of the more established Macintoy programs such as PageMaker, from Aldus Software. An excellent move by Gold Disk was to avoid copy-protection, and keep the price down to a very reasonable $300, considering the power of the program and the going rate for such programs of around $1000.

In a very few minutes I was on the road, generating text boxes (the basic unit in ageSetter). A series of boxes can be chained together to make an "Article;" Articles can be spread over an arbitrary number of pages, and you can easily pop from one box to another of an article with a click on a gadget.

One thing I find very hard to understand is the lack of an Undo function on the page formatter. All the other modules have it. The best it does it to give you a second chance to stop it (do you really want to...). When you say yes, you say goodbye to your work. To quote Pink Floyd, you have no recourse to the law...

PageSetter has four main modes: The page formatter (the main module), the Text Editor Department, the Graphics Editor Department, nd the Press. The purpose of each of these is obvious, and the other three are available from the "Department" menu of the page formatter.

When formatting a PageSetter page, you drag out boxes (like in a painting program ... such as the one in the "Graphic Editor Department") of the required size, then fill them with text or graphics. It will only allow one or the other, and only one font per box, though you can overlay them.

PageSetter comes with a "flyer" document for you to play with; this is a good touch, but it would have been polite to tell you where it is! It is in fact in the folder Empty/Documents. I had to exit the program and hunt around with the CLI to find it. The file selection boxes do not exclude non-pagesetter files, meaning that you have to sift through a large list of files to find the ones you want, and since it does not "cache" the directory, every time you load a new file, it has to re-read the directory.

Text Editor Department.

This is a fairly standard sort of screen editor, which can import plain text files, or files from Scribble or TextCraft. Rather than being WYSIWYG (What-you-see-is-what-you-get), it uses control sequences like \b for bold and \i for *italic*. It has a scroll bar for moving around within a file, and the usual text-editing functions such as search-and-replace.

Once you get used to it, it is a quite responsive and no-nonsense editor. It would be nice if it were WYSIWYG.

There is a gadget on the main screen called "QuickText" which is a tiny version of the Text Editor, for entering such things as headings. The manual says that it is limited to about 100 characters.

Graphic Editor Department.

This is a bare-bones painting program, which is quite adequate for the uses for which it was intended; It comes with a considerable library of "ClipArt" on disk; There is one slight problem with those files ... they are too densely packed on the screen, so you have to erase parts of the pictures around the one you want before you can "cut" the picture out.

The Press.

The Press is the most important part of PageSetter, and unfortunately, the weakest part also. It is limited to the resolution of the Amiga fonts that it prints; this means that no matter how good your printer is, there is a fairly low limit to how good the final product will look.

It also has very few controls; when you have started the print job, there is no way to stop it. In fact, the only way I found to stop it was to put the printer off-line; a few minutes later, it complained and stopped trying.

Little Annoyances.

It is often said that reviews are boring unless the reviewer has a few axes to grind...

When I started experimenting with PageSetter, I first copied my System-Configuration Preferences to the disk. When I attempted to print my document later, it told me "Printer Preferences Not Appropriate". It didn't say what was wrong ... just that something was wrong. As it happens, it was complaining about the darkness threshold which was 3 instead of 2.

It would have been thoughtful to form-feed the printer after printing a document, and to allow the user to abort printing a document. As it is, the only way to stop it is to put the printer off line and wait about a minute for the "Printer Trouble" requester to appear, then wait another 20 seconds.

The manual has no index, and has an annoying habit of making "forward references" (telling you that something can be done without telling you how, or where to find out). An example of this is the "Autobox" reference on page 2-14. After reading the manual carefully, I re-read it trying to find this entry, which is another "what it does" rather than "how you do it" entry. Essentially, it does it all by itself: formatting the page into columns automatically

The manual also has some subtle errors to fool the novice, such as "Inserting a '-' into a word tells PageSetter where to hyphenate..." when they mean '\-' or 'Soft Hyphen'.

The manual is also very short on examples of any kind, mainly screen images annotated with labels to describe the gadgets, and so on. It has a very useful glossary, but it has a wordy explanation of such things as bold and *italic* where a tiny example would have sufficed.

There is an intriguing bug in the handling of the screen colours; One of the files I have resets the screen colours to a very sick combination of (I think) the colours it came up with. Strange. This means that the authors are playing with the Preferences colours, which is a big no-no. (It uses the WorkBench Screen.)

The requesters (and other parts) were obviously written by several different people; there is a big consistency problem there. If you get the chance, have a look at the requesters in the various modules; they are nearly all different, where they could be the same. (It would even save space). The biggest flaw in this regard is the font selector in the box menu; rather than scrolling around, or allowing the mouse to select, the "current" line is the top of the list. Every time I go into this requester, I have to rethink. None of the requesters in this program understand "double-clicking". You have to select something, then use th "OK" or "Load/Save" gadgets, or whatever.

The "mop" gadget on the main screen says that it will "mop up" (delete) the current box. When you OK this operation, it instead deletes the whole article, (ie all boxes in the current chain).

Nit-Picking...

The file requester in the Graphics Editor can be "lost" irretrievably by selecting its window-to-back gadget. Most of the other requesters in PageSetter do not have window front/back gadgets, so this is not a big problem. In the main screen, though, you can get it back, since the main window ha front/back gadgets. From the Graphics Editor, there is nothing you can do but reboot, since even the menus are disabled during the file request.

On the subject of requesters, try the following trick on the main screen: select the colour/paper entry on the menu bar, then click on the Grid gadget. You will be as surprised as I was with what happens ... the gadget changes state without doing anything! There are several other ways to do this. Minor point, I know.

I just *wish* it was possible to mix fonts in a box. It seems silly to have to go to such lengths to get more than one font ... and even if you do manage it, it will likely fail as soon as you change box dimensions or insert some more text.

Redeeming features.

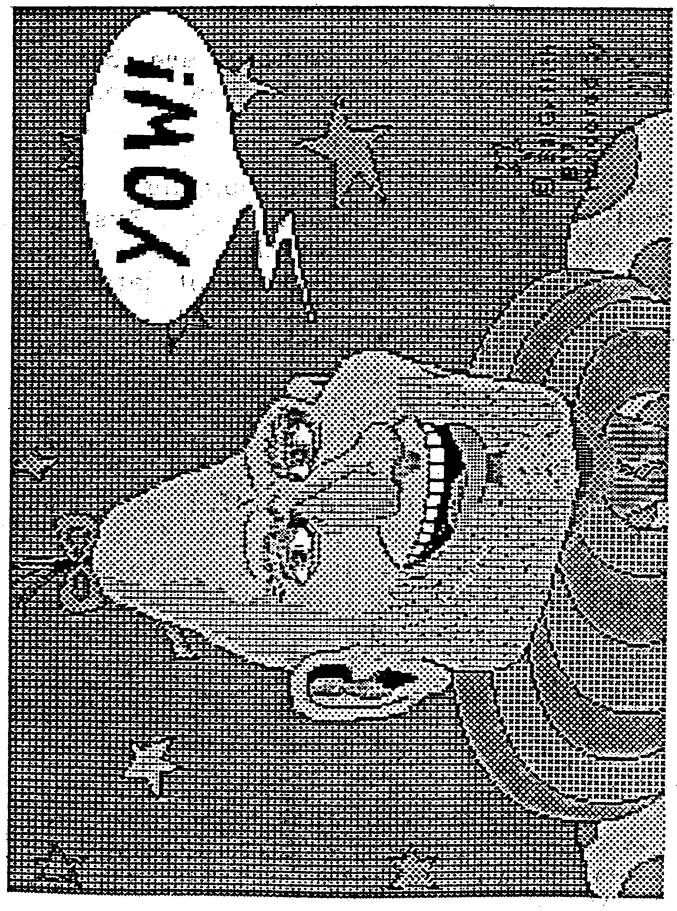
The manual does provide some excellent advice on document preparation and how to organise yourself. It also has a very useful glossary of printing terminology, but their definitions tend to be wordy and lacking in examples.

QuickMove is another excellent feature. Normally, when you relocate a box, there is a delay of a second or two before it will move (just like the Atari always acts...); with QuickMove, only the box moves, and it *is* quick. I recommend this mode, and wish it was the default.

Summary.

In spite of the above criticisms, I like PageSetter. It is quite an impressive program, and a long way towards the ideal of Desktop Publishing.

If you want to get into this field, *now*, and not in a few years' time, I can certainly recommend it, since it is the only program of its type available for the Amiga!



This was printed on a Hewlett-Packard LaserJet, (Photoreduced) using DeluxePaint.

HELPFUL HINTS: Reprinted from Amiga News

As you all know (don't you?) you can assign logical volume/device names to physical disk drive names such as:

```
assign Source: df0:
assign Destination: DF1
diskcopy source: to destination:
```

Useful?, yes, for some things.

Did you know you can also assign logical volume/device names to sub-directories? Such as:

```
assign from: df0:devs/printers
assign to: ram:
copy from: to to:
```

BUT the *FUN* one you may not have noticed is that you can assign to FILE NAMES (read commands). For example:

```
assign x: c:execute
assign e: c:ed
assign cc: df1:c/LC
```

then use them like:

```
x: startup-sequence
e: my_text_file
cc: foo -i df1:include
```

Assigns are SYSTEM WIDE, NOT unique to each window.

Have Fun!

Bruce Barrett
Commodore-Amiga

Tale of two computers.

Having the fortune (good or bad) to work with PC's (among other machines), and having an extensive library of software designed for them, I was in a position to easily justify buying a PClone in addition to my Amiga.

Being in somewhat of an adventurous mood, wanting a hard disk on my Amiga, and having decided to buy a clone of some sort, I decided to give SideCar a go first, and see if it lived up to the expectations everyone had for it, since it promised to solve all my problems in one fell swoop.

So in late December, I made my way out to Steve's and had a good look at SideCar...and decided that it would be an interesting machine to own; I came home that day with a SideCar and a 20Mb HardCard (amazing piece of technology: a 20M Disk and controller on a single card...and virtually indestructible).

It seemed like a good idea to leave the HardCard out while I checked out the SideCar. It worked perfectly, first time. Scrolling was a little slow, but you can't have everything. So I added the HardCard, which meant unplugging everything and unscrewing the top (the box is cunningly designed so that you can't take the top off while it is plugged into the Amiga).

It was a damn close fit, and I thought at one stage that I was pushing *too* hard, but it just would *not* go into that slot. All the cables went back together and I turned it on; I had been warned that it *might* take over ten minutes to boot with the HardCard, and to reformat the Hard Disk with a different interleaving factor. I did not expect to have to wait 15 minutes, though. Not to *boot!*

Interleaving is a way of improving disk performance by staggering the sectors on adjacent tracks; the idea is that since the disk takes a certain amount of time to spin around, and a certain amount of time to "step" the head from track-to-track, when the head *does* move from one track to another, the chances are, that when it gets there, it hasn't just missed the sector it was looking for. In the case of Amiga, a full track is read each time, anyway.

We can rebuild it...

So eventually the machine booted and I ran the program that Commodore-Amiga supply for this very reason; it is called *hdf1.* It would be a nice touch to translate the messages and prompts from German into English...

An apparent eternity later (about half an hour), the program completed, and I ran its sister program, *hdf2,* which validates the newly formatted disk. This ran with no errors. Ok, that's that. Now to partition the disk.

In addition to the normal "FDISK" program, Commodore-Amiga supply a program "ADIS" to set up the Amiga Partition. SideCar allows *both* machines to use the same hard disk, through some combination of silicon and black magic.

Each of these programs took about two minutes to run, but each rebooted the machine, which meant another fifteen minute wait...sigh.

Formatting the thus generated partitions was relatively quick; the MSDOS partition was boring, as usual, but on the Amiga side, there is a new program called *DJFormat.* It looks just like *Format,* but runs slightly quicker, and in my case, went out to cylinder 460-odd. When it was formatted, I copied the 1.2 distribution disk onto it. In light of the format time, I was surprised at how slow it was; slightly faster than copying to a floppy disk. Ah well, can't have everything.

Some programs have the device name "HD0:" in their file requesters, and not "DJ0:"; this is easy to fix. All it takes is a simple *assign* statement in the *startup-sequence* file.

Next step was to make a boot disk; Commodore-Amiga supply a program to do this, too. That was another 15-20 minutes. Setting up a SideCar HardDisk is a series of simple commands and long waits.

I started this job at 5.30pm, and it was now 2am. Time to put the cover back on and call it a night. I turned everything off, pulled it apart, and screwed it all back together.

When I had bolted the whole thing back together again, I checked that it was still working before I hit the sack for the night. Somehow, I got tied up in it again, and did some more "customisation". About half an hour later, I had a very nasty shock. While in Preferences, changing the mouse pointer, the screen folded, just as if I had turned off the Amiga.

Herein lies a story.

Tale of no computer...

In the manual, there is a paragraph in bold type that says something to the effect *if the Amiga is off when the SideCar is turned on, extensive damage will be done to the Amiga.*

Wonderful. When I read that, I thought it was an overstatement. SideCar has a short power cord that plugs into the power socket of Amiga. I turned everything off, unplugged SideCar, and noticed that the power cable did not seem terribly tight. I tried turning on the Amiga again, in the normal way. Whew! Kickstart requester. I put the KickStart disk in...nothing happened!

Certain that the impossible had happened, I dug out the boxes, packed it all into the Car and had a restless night's sleep.

First thing next morning, I was out at Steve's. Before I knew what was happening, I had a new Amiga under my arm. I told them that I never wanted to see that SideCar again, after what it did to my Amiga, and we made an arrangement, whereby I am now the owner of an Epson PC instead.

This demonstrates the *best* reason to do business with a big dealer. When it came to the crunch, they were there, and cooperative. Thanks, guys.

The moral of the Story...

The fact that my Amiga was Brain-burned may *or may not* be related to that fateful passage in the SideCar manual. It would not be the first Amiga that I have heard of spontaneously dying.

The fact that it happened within eight hours of plugging in a SideCar *is* what the point is. Also that the manual *states* in no uncertain terms what will happen if someone turns off the Amiga while the SideCar is on.

All it would take would be a Amiga owner *without* a SideCar to decide to reboot under a different operating system, just like they do at home...and no more Amiga. Or someone to bump the table (which seems to be what happened to me.) That is not sensible, that is damn stupid. It ruins what is potentially a very good machine. You shouldn't *have* to worry about which machine you flip the switch on first.

For the sake of a few dollars worth of interface chips, Commodore-Amiga have forsaken their customers, and left themselves with a market-ful of time bombs.

Frankly, I am not willing to live with that sort of albatross around my neck; That's why I now have a PClone instead, even though it gets high marks for compatibility.

My experiences also changed my mind about hard disks on Amiga. I no longer want one. I have tried to fill one. I managed about three megabytes, and ran out of things to install there; in fact, I deleted a bit of that!

The basic problem is *what* do you put on it? My disk collection is rather large, and I find it more convenient to reach for a disk than for a directory. It is also easier to back up.

The Hard Disk is quicker, but not *that* quick. Also, there is no backup utililty for Amiga, so if something *should* go wrong, you're faced with a long haul reloading files.

A RamDisk is much faster. I have used RamDisks extensively, and *know* that they suit my working environment; you load what you *need.* With the RESIDENT mode in Workbench 1.2, you can even make programs load instantaneously. Now *That* is progress.

As a machine, SideCar is good; it is also a great risk. Doing business with a big dealer made all the difference when it came to the crunch, and I am now quite happy with my setup.

---oOo---

WHAT'S ALL THIS ABOUT MIDI, THEN ?

Midi (*Musical Instrument Digital Interface*) is a language which has been created to control and synchronise one or more musical intruments or pieces of electronic equipment. For example, *MIDI* may be used with **musical keyboards, drum machines, guitars** (with the appropriate interface), **lighting controllers** and **computers** such as the *Amiga.*

The Origins of *MIDI*

The first keyboard synthesizers were primitive monophonic devices (could only play one note at a time) which must have been rather frustrating to use, especially if a "full sound" was required by the musician. An attempted solution to this problem was found by allowing two monophonic synthesizers to be **slaved** together electronically, so that playing a note on one keyboard would cause the second keyboard to play the same note. This concept used *Voltage Control* to instruct the slave keyboard which note to play at any moment. Basically, when a key was depressed it sent out a voltage to the synthesizers *voltage-controlled oscillator* (VCO) which generated a tone, depending on the voltage applied to its *control voltage input.* Most VCO's followed a one volt per octave response, so that effectively the pitch of the tone created **doubled** for each voltage increase of 1 volt. Hence, by allowing the control voltage from one synthesizer to be passed to a second (the slave) the second keyboard (with perhaps a different sound) could be played automatically.

There was one other problem which needed to be solved before master/slave arrangements could function properly. This problem was caused because synthesizer oscillators generate tones *all of the time.*
Once you hit a key and generate a control voltage, the keyboard will keep sending that voltage until another key is hit (or you turn off the machine). For the master keyboard this is handled by a second circuit called a *voltage controlled amplifier,* which gates the oscillator tone on and off, depending on whether the key is still being held down. Thus if we make sure the master keyboard has a gate output and the slave has a gate input, the two synthesizers can be controlled together. The arrangement between these devices can be seen as a primitive analog language, which hinted at the possibilities of further control given a more powerful digital language.

The Problem of Synchronisation

The master/slave relationship worked fine for combining keyboards, however once drum machines were introduced a new problem arose. Musicians wanted to be able to synchronise the beats created by a drum machine with a keyboard sequence, thereby creating the types of sounds we are familiar with today. These devices used a clock to keep time (ie. when to play the next note of the sequence or drum sound) and some means of combining these devices was required.

The solution is similar to the master/slave keyboard arrangement, except rather then sending out VCO/VCA information the master unit (usually the drum machine) sends out its clock signal as timing for the slave device (the synthesizer). So it came to pass that most electronic musical equipment possessed voltage, gating and timing information which could be generated towards other devices, and the ability to select from an internal or external signal (master or slave).

A Brief View of MIDI

We now know some of the problems associated with electronic musical equipment, so lets look at how **MIDI** deals with these problems and its basic form.

The MIDI Language

All *MIDI* communication is acheived through multi-byte *messages* consisting of **one** status byte followed by zero or more **data** bytes. These messages are sent between electronic devices at a serial rate of 31.25 (+/- 1%) Kbaud, asynchronously, with a start bit, 8 data bits and a stop bit. The power of *MIDI* comes from the ability of a digital language to carry a far greater range of information then the analog counterparts we saw earlier.

The most basic type of information the *MIDI* language conveys concerns notes - which notes are being played and for how long. For example, to indicate that a note on a particular channel (there are sixteen different *MIDI* channels to send information down) has been hit a status byte indicating the channel number would be generated, followed by a data byte indicating which of 128 notes has been depressed (a value in the range 0 - 127 , with middle C = 60). *MIDI* also transmits *timing and synchronisation data,* so that drum machines, sequencers, and keyboards can all start, stop and play together. *Program changes* are also part of the information package, so that if you select a particular program (sound) on one synthesizer, any slave synthesizer will change programs, thereby complementing the sounds coming from the master synth.

This information is physically carried between machines using a *MIDI* cord, which is typically a five pin DIN connector.

At present the *MIDI* signal requires only three of the five terminals for operation, although there is speculation that the other two terminals will be used someday as *MIDI* evolves. As we saw for our analog counterparts, we require an output (from the master) and an input (to the slave) to pass this information.

MIDI, however, requires a third type of port for communication - the **MIDI THRU** port. The reason for this becomes obvious when we consider linking more then one slave device together from a single master. *MIDI* information is generated from the keyboard or controls of the synthesizer as they are used, so that the master keyboard can send information to the slave via its **MIDI OUT** port. The slave reads this information from its **MIDI IN** port, however, because no physical changes occur to the slave device, no *MIDI* information is generated on the slave **MIDI OUT** port.

For a third device to be controlled from the same master unit the *MIDI* information must be passed through the slave and onto the next slave. The **MIDI THRU** port serves this purpose, basically passing the master commands unchanged out of the **THRU** port, so that further slave devices may be controlled. This technique is called "daisy-chaining".

So we now have the situation whereby different electronic devices may send serial data to each other, allowing the control of notes to be played, effects to be created and the ability to synchronise all of these operations by a common timing language.

So What Has This Got To Do With The AMIGA ?

MIDI information is nothing more then a string of serial bytes which convey messages to control aspects of various devices which are MIDI compatible. The *Amiga* is quite well suited to creating these bytes of control information, as it looks like any other type of binary information which a computer is happy to deal with. To use the *Amiga* with your electronic equipment you require two basic ingredients :

1/ **A Midi Box** - converts serial data from the *Amiga* serial port into data compatible for transmission down a *MIDI* cable.

2/ **Software** - a program to create or store this data

I have used the **PIECES SIM-102** Amiga midi interface which appears to offer all of the requirements for a **Midi box,** with **MIDI IN, MIDI OUT** and **MIDI THRU** ports.
This interface, which connects neatly to the back of the *Amiga* via the serial port, will work with all MIDI compatible software which uses the *Amiga's* serial port, including **SoundScape, Music Studio** and **Harmony.**

So, the power of a digital computer can now be used to control devices which are traditionally analog in nature, and the scope for future directions of software and hardware usage for this interface seem to be just beginning.

NOTES AND ERRATA.

Yes as from issue one we have made mistakes! Don't worry, they weren't too bad.... well?

Adding disk drives. (Diskchange)

This article contained incorrect statements about the DISKCHANGE signal. This signal is latched by the drive UNTIL the drives heads are stepped. This means that the signal is cleared when the heads are stepped and the drive is selected, the Amiga then checks the drive for the loading of another disk by checking for valid data etc. It is not latched until the heads are stepped AND there is a disk in the drive, as stated in the article. If you have tried the circuit in the article no damage will have been done, the only consequence of this would be the need to have disks in all other drives before the disk change would be recognized.
(Peter McNeil)

(See the circuit diagram on the next page)

SENDING ESCAPE CODES IN AMIGABASIC

Reprinted from Amiga News

by Carolyn Scheppner, CBM Amiga Technical Support

As you've probably realised, AmigaBasic eats most console ESC sequences and instead gives you the little box.

But you CAN send the standard ISO escape sequences to the printer from AmigaBasic (See RKM (Rom Kernel Manual) Vol.2 printer.doc for ISO sequences). Basic's printer device (LPT1: which is used by LPRINT) is flakey. It tends to munch on initial escape sequences. If you must use LPRINT, you can often get the ESC sequences thru by printing something (like a space) prior to the sequence.

But there's better way. Open a PRT: file and PRINT# to it. This example prints "hello" in NLQ (Near Letter Quality) mode and then in normal mode.

(Note: DEN2 (NLQ on) = ESC[2"z DEN1 (NLQ off) = ESC[1"z)

```
OPEN "prt:" FOR OUTPUT AS #4
esc$ = CHR$(27) + "["
den2$ = esc$ + "2" + chr$(34) + "z"
den1$ = esc$ + "1" + chr$(34) + "z"
test$ = "hello"
PRINT#4, den2$; test$
PRINT#4, den1$; test$
CLOSE 4
```

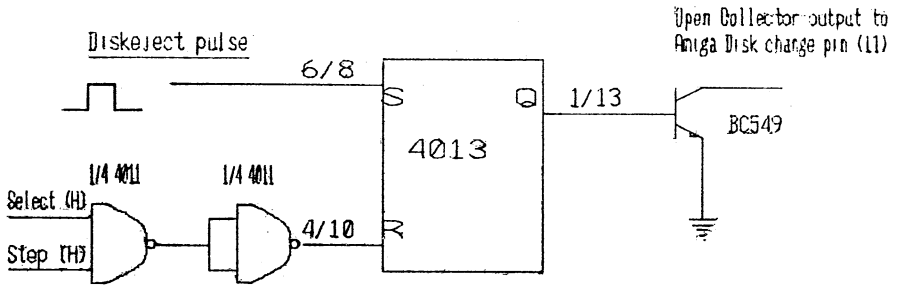
Of course, your printer has to support NLQ for this to work. Other sequences are possible, see the RKM for a list of printer control sequences.

You can test the sequences from CLI. Try typing the following. Everything typed between the first and last line will go to prt:. (Note: <RET> is the return key, <ESC> is the escape key)

```
copy * to prt:<RET>
<ESC>[2"zhello<RET>
<ESC>[1"zhello<RET>
CTRL-\                          (Note - Hold CTRL and type \)
```

With my MPS 1000, this prints "hello" twice, first in NLQ then normally.

(Correction to diagram in last issue)



Disk change circuit.
This one should work!

NB. The disk pulse should be a PULSE,
To set the 4013
To make a pulser use a monostable circuit
triggered by the disk present switch.

BECAUSE

36 Ambalindum St.
HAWKER A.C.T. 2614
Ph. (062) 54 6033

Bulletin of the Canberra Amiga Users Society